So in this lecture, I'm going to show you how to refactor some legacy code, and make it testable and loosely coupled. So in a project, look at this mockup folder.

Here we have this class video service, look at this method. Read VideoTitle. So here, you read the content of the file called video.txt, and store it in a variable called str. Right? And then we give it to this json convert to deserialize this string as a json object. If you have never worked with json, json is basically JavaScript object notation. So in JavaScript, we have a special notation for objects using key value pairs.

You're going to see that in a second. All that we're doing here is passing a string, and we're expected to get an actual video object.

I've defined this video object right here, so, it's a class with three properties. ID, title, and IsProcessed. Okay, so back to our method, as part of converting that string with json object something may fail, maybe that string is not in the right format. In that case, we are not going to get a video object, we're going to get null and will simply return an error message to the client of this class.

Otherwise we're going to return the title of the video. So a very simple method, but on the first line, we're touching an external resource, in this case a file system. So we want to do some surgery here, and decouple this code from the file system. As I told you before, the first step is to move all the code that touches an external resource into a separate class and isolate it from the rest of the code. So in this case, I want to move this expression into a separate class. So, back in the project, in the mocking folder, I'm going to add a new class.

I'm going to call this class FileReader. It's responsible for reading

a pdf file. So FileReader here we need only one method for reading the content of a file,

So public, string, Read perhaps it gets the path of the file.

Now, back to our video service, I'm going to cut this code from here, and paste it here. Now, let's import the name space for the file class.

And also replace video.txt with this path argument. So we can reuse this class in multiple places. And finally return the content. So now, the code that touches an external resource is encapsulated and isolated inside this FileReader, okay, now, back to our video service, here I'm going to new up a FileReader and then call Read and pass video.txt here. So we haven't changed the behavior of our application, the code still being the same thing, but we simply restructured or refactored our code. Now, in this implementation, this video service class is tightly coupled to FileReader, because we are newing up an instance of FileReader here.

And that means in our unit tests we can not replace this FileReader with a fake object, or a test double. So what should we do here? We should program a against an interface. So instead of working with a concrete implementation, like this FileReader, we should work with an interface or a contract.

Then, at run time, we can pass any object, that implements that interface. Okay? So the second step that you need to do here is to extract an interface from the FileReader class. So back to the file reader.

I'm going to declare an interface. Public interface all interfaces in .net should start with i, and then call this FileReader. Now as I told you before, interfaces only have declarations.

They don't have any implementation. So in this interface we want to

declare a method with this signature, a method called read. That takes a string, and returns a string. So I'm going to copy this and paste it here. And by the way, members define in an interface don't have an access modifier, like public or private. Because an interface defines a contract. So when a class implements an interface, it's going to define all the members declared in an interface, as public members. Okay? So, here's our interface. Now, in order to tell the C Sharp compiler that this FileReader implements this interface we add colon IFileReader. So this is how we extract an interface from a class manually. But let me show you a faster way.

So, I'm going to delete this part here, I'm also going to delete the interface declaration. If you're using Visual Studio, you can right click on your type and go to the refactor menu. Here, I'm using Rider so I have to press Alt and enter, and this is refactor menu. When you go here, we should see an action called extract interface.

So here, Rider is asking me what is the name of this interface, by default it suggests IFileReader. Because it follows the convention in .net. Next it's asking me where I should place an interface, should this be next to this class, or should this be in another file. For now it's okay to put this next to this class. In more complicated enterprise like applications, you may want to put an interface in a separate file, or even in a different assembly.

That's beyond the scope of this course, so for now, don't worry about it. And finally, we need to select the members that should go in the interface. So here, I'm going to select the read method. Next, Look, we have this interface here, and our FileReader class implements this interface. So here's one implementation of this interface where we use the file class to read the content of a file in the file system, now that we have this interface or this contract, we can also create a fake implementation that we use in our unit tests. So, back to our solution explorer, in our unit testing project I'm going to add a new class. New Class. And call this FakeFileReader. Now in unit testing, we actually have two different types of fake objects. You might have heard of stubs and mocks.

They have some similarities, but they have the subtle difference they both represent a fake object. Some older unit testing frameworks differentiate between stubs and mocks, but more modern frameworks don't differentiate, and I find this more practical. So here we can call this class fake FileReader, or we can call it mock FileReader, or stub FileReader, all these names are perfectly fine.

So I want to call this fake FileReader. Now, I'm going to have this class implement our new interface. IFileReader. And this is defined in testninja .mocking name space. We have this compilation error, because we have not implemented the members declared in this interface, so we press alt and enter, and select implement missing members.

So now we have this read method in this class, but instead of using the File class in .net, I'm going to return a hard coded string. Something like this, so I'm not accessing the file system, this is a fake implementation. And we can use this in our unit tests. Also, note that I have defined this class in our unit testing project, so it's not part of our production code.

Okay? So let's quickly recap what we have done so far. In video service, we extracted the first line, into a separate class called FileReader. Here's the implementation of our FileReader class, and we also extracted an interface from this class called FileReader. And finally we created another implementation of this interface called FakeFileReader and this is the one that we're going to use in our unit tests,

Now there is still one step left. Back to our video service, our video service class is still tightly coupled to this FileReader class. But as I told you before, here, we should program against an interface, in this

case IFileReader. So you shouldn't really new up this FileReader object here. Now there are three ways we can pass an instance of a class that implements IFileReader interface with this video service. One way is to pass it as a parameter to this method, another way is to pass it using a property, and the last way is to pass it by a constructive parameter.

For the next three lectures, you're going to see each of these approaches

in action.